Greenbelt MD 20771
March, 1987

Abstract

In object-oriented software engineering, the software developer attempts to model entities in the problem domain and how they interact. Our previous work, which grew out of a Goddard Space Flight Center Software Engineering Laboratory Ada (tm) pilot project, has concentrated on using object-oriented ideas in software design and implementation. However, we have also found that object-oriented concepts can be used advantageously throughout the entire Ada software life-cycle. This paper provides a distillation of our experiences with object-oriented software development. It considers the use of entity-relationship and object data-flow techniques for an object-oriented specification which leads smoothly into our design and implementation methods as well as an object-oriented approach to reusability in Ada.

Introduction

The Goddard Space Flight Center Software Engineering Laboratory is currently involved in an Ada (tm) pilot project to develop a system of about 50,000 statements.[18] This project has provided both experience in using Ada and acted as a testbed for new Ada-oriented software development methods. The system, a satellite attitude dynamics simulator is based on the same requirements as a FORTRAN system being developed in parallel.

Increased productivity and reliability from using Ada must come from innovative application of the non-traditional features of the language. However, past experience has shown that traditional development methodologies result in Ada systems that "look like a FORTRAN design" (see, for example, Basili, et. al.[3]). Therefore, for the pilot project we decided to begin with the system requirements and redesign the system, exploring new design strategies.

Unfortunately, the system requirements given to our team were highly biased by past FORTRAN designs and implementations of similar systems. Therefore we began by recasting the requirements in a more language-independent way using the "Composite Specification Model".[1] This method involves the use of state transition and entity-relationship techniques as well as more traditional data flow diagrams. We then designed the system to meet this specification, using object-oriented principles. The resulting design is, we believe, an improvement over the previous FORTRAN designs.[2] The system should be coded and tested by the summer of 1987.

Previous work by the present authors has concentrated on using object-oriented ideas in software design and implementation. This work resulted in a design method which synthesizes the best methods studied during the pilot project.[23,24] However, we have found that object-oriented concepts can be used advantageously throughout the entire Ada software life-cycle. This paper provides a distillation of our experience with the pilot project and other Ada projects into an evolving life-cycle methodology in continuing use at Goddard.

Specification

The modules of an object-oriented design are intended to primarily represent problem domain entities, not just functions. Therefore, proceeding into object-oriented design from a structured analysis requires an "extraction" of problem domain entities from traditional data flow diagrams.[5,6,23,24] From an object-oriented viewpoint, it seems appropriate to instead begin a specification effort by identifying the entities in a problem domain and their interrelationships. Entity-relationships and data flow techniques can then complement each other, the former delineating the static structure problem domain and the latter defining the dynamic function of a system. This is similar to the "contextual" and "functional" views of the Composite Specification Model.[1]

Entities and Relationships

An entity is some individual item of interest in the problem domain. For example, consider the specification of a system to simulate the dynamics of a spacecraft in Earth orbit. Several problem domain entities immediately come to mind: the spacecraft, Earth, thrusters on the spacecraft, etc. An entity is described in terms of the relationships into which it enters other objects. A spacecraft might be in a certain orbit state, have certain thrusters, etc. Entities can also have attributes, such as spacecraft mass, which are effectively simple relationships with standard data items.

The following is an example of the possible relationships into which "spacecraft entities" might enter:

Ada is a trademark of the US Government (Ada Joint Program Office).

SPACECRAFT
Parameters SPACECRAFT PARAMETERS
State ORBIT STATE
Thrusters { THRUSTER }
Computer ON-BOARD COMPUTER

ORBIT STATE
Position INERTIAL VECTOR
Velocity INERTIAL VECTOR

THRUSTER
Parameters THRUSTER PARAMETERS
Firing BOOLEAN

ON-BOARD COMPUTER
Uses { OBC DATA VALUE }

This indicates, for instance, that the "state" of a spacecraft is an "orbit state" which has a "position" and a "velocity". Curly brackets indicate a relationship with "zero or more" of the bracketed entity.

The entity-relationship diagram (ERD) is a common graphical tool for entity-oriented specification.[9] Figure 1 shows an ERD for the above entities. The notation for this diagram is from Martin and McClure.[14] This diagram shows only a small part of the example problem domain. It would grow as additional entities and relationships are added to describe additional parts of the problem domain. As the specification grows, a complete ERD can quickly become cumbersome. Therefore a textual "entity dictionary" seems to be most useful as the primary entity specification, with ERD notation being a graphical way to map parts of it.

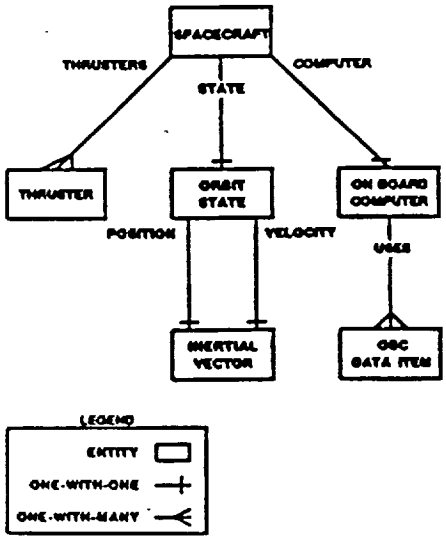

FIGURE 1 Orbit Simulator Entity-Relationship Diagram

The goals of system specification are actually quite similar to the goals of knowledge representation work in artificial intelligence. Therefore it is not surprising that there are similarities in techniques. Entity-relationship diagrams are basically the same as "semantic networks" in AI,[21] and the entity dictionary is similar to the AI concept of a "frame".[17] These and other AI knowledge representation techniques may be increasingly applicable for complicated system specification. This also suggests the intriguing possibility of developing a sophisticated "specification assistant" system which directly "understands" entity relationship specifications.

Object Data-flow Diagrams

ERDs show all possible relationships between different types of entities. They do not show the actual relationships between specific entities at specific points in time, nor how these actual relationships change over time. Data flow techniques, however, provide exactly this dynamic view. Traditional data flow diagrams (DFDs) show the flow of data between functional processes. We will, instead, diagram the flow of data between objects which represent specific parts of the problem domain. This results in object data-flow diagrams (ODDs) for the dynamic view of the specification.

For the specification phase, objects are not meant to be software modules, but to represent the dynamic view of one or more entities in the problem domain. A specification object is effectively a state machine which accepts input data, processes it and produces output data, possibly modifying some internal state data. It has no "operations" as such, only data flowing in and out. In this ways ODDs are similar to Buhr's "cloud diagrams",[7] though ODDs are oriented towards specification rather than design.

To construct an ODD specification, one needs to identify the main entities involved in dynamic processes. In the case of the orbit simulator example, the function is to update the spacecraft state in response to environmental forces and thruster firings under control of the on-board computer. We thus specify a DYNAMICS object to represent the physics of motion of the spacecraft, a THRUSTER SIMULATOR object to represent the firing of thrusters and an OBC EMULATOR object to emulate the operation of the on-board computer.

Figure 2 shows the data flow between these three objects. Note that each datum flowing on an ODD is itself an entity. The entity dictionary can thus completely replace the traditional data dictionary. Some of these entities, such as "thruster status" and "OBC telemetry", may actually not be identified until the dynamic view is considered. Thus the entity dictionary will most likely continue to grow and be refined as the ODD specification is constructed.

An ODD specification must include a detailed specification for each object which appears on an ODD. An object specification provides a statement of the problem domain abstraction represented by an object. It should include a textual description of the object as well as a listing of all inputs and outputs. The object specification also provides a place to include "non-

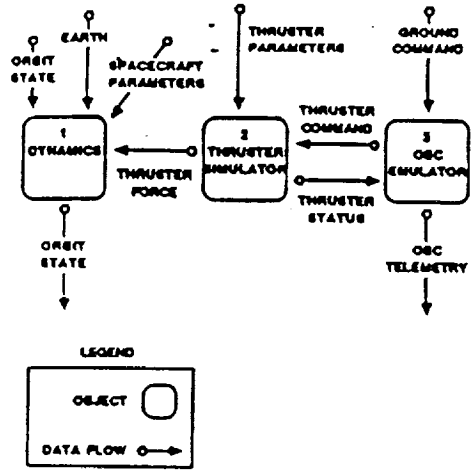

FIGURE 2 Orbit Simulator Object Data-flow Diagram

functional" requirements such as timing and accuracy constraints.

The object specification must also detail the function of the object. This could be in the form of structured English, a state transition diagram or some other appropriate notation, such as differential equations for the evolution of an "orbit state". The function of an object can also be given by a lower level ODD. Decomposition can continue recursively on all ODDs until all objects have been decomposed into primitive processes and states. This results in a leveling of ODDs similar to the leveling of traditional DFDs. However, unlike DFDs, each object at each level of an ODD specification has a complete object specification. Each object should also represent a good problem domain abstraction independently of its decomposition.

An ODD is a specification tool showing data flow rather than software control structuring. However, control issues can actually be included on an ODD when absolutely necessary. Larger arrows without circles on their tails indicate signals from one object to another. An ODD specification should include the minimum such "control flow" absolutely necessary to specify a system. With this added notation, ODDs are effectively the same as the (unannotated) "process graphs" used in PAMELA (tm).[10]

Design

A system specification describes what a system should do in terms of the problem domain. The main task of design is then to impose a control structure on the system function to allow software implementation. In object-oriented design the unit of modularity is the object, this time considered in the usual sense of a packages of data and operations on that data.[4,13] Ideally, the objects in the design should directly reflect the objects in the specification. However, various design considerations may require certain specification objects to be grouped together or split apart to construct design objects. Further, there will almost always be additional objects in the design to handle "executive" and "utility" functions.

Designing with Objects

The intent of an object is to represent a problem domain entity. The concept of abstraction deals with how an object presents this representation to other objects.[4,12] During specification we deal with objects with high abstraction, close to the problem domain. In design, however, there is a spectrum of abstraction, from objects which closely model problem domain entities to objects which really have no reason for existence.[23,24] The following are some points in this scale, from best to worst

Entity Abstraction - An object represents a useful model of a problem domain entity.

Action Abstraction - An object provides a generalized set of operations which all perform the same kind of function.

Virtual Machine Abstraction - An object groups together operations which are all used by some superior level of control or all use some junior level set of operations.

Coincidental "Abstraction" - An object packages a set of operations which have no relation to each other.

The stronger the abstraction of an object, the more details are suppressed by the abstract concept. The principle of information hiding states that such details should be kept secret from other objects,[4,19] so as to better preserve the abstraction modeled by the object.

The principles of abstraction and information hiding provide the main guides for creating "good" objects. These objects must then be connected together to form an object-oriented design. In contrast to the data flow orientation of ODDs, our object diagram design notation[23,24] shows control flow and module dependencies between objects. This software structure must, however, preserve the specified functions and necessary data flow, though the actual data flow paths may be altered.

Design Hierarchies

The transition from ODD specification to object diagrams is mediated by consideration of two orthogonal hierarchies in software system designs.[22] The composition hierarchy deals with the composition of larger objects from smaller component objects. The seniority hierarchy deals with the organization of a set

PAMELA is a trademark of George W. Cherry.

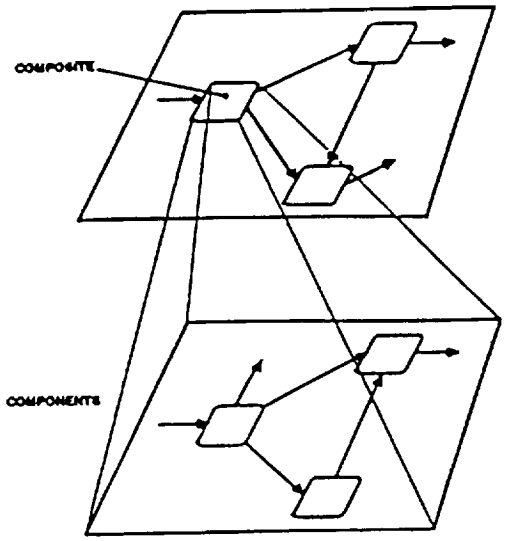

FIGURE 3 Composition Hierarchy

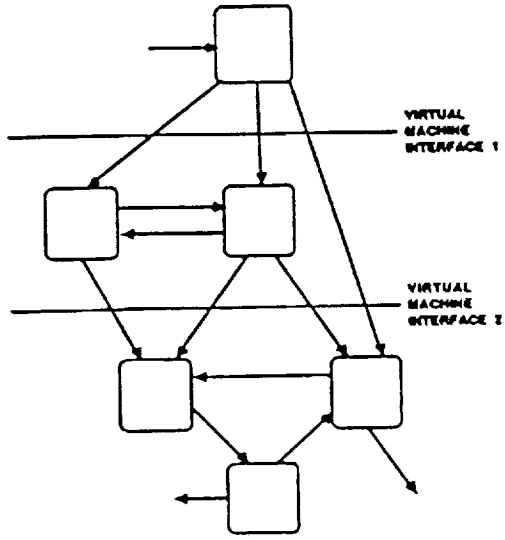

FIGURE 4 Seniority Hierarchy

of objects into "layers". Each layer defines a virtual machine which provides services to senior layers.[12] A major strength of object diagrams is that they can distinctly represent these hierarchies.

The composition hierarchy is directly expressed by leveling object diagrams (see figure 3), similar to the leveling of ODDs. At its top level, any complete system may be represented by a single object which interacts with external objects. Beginning at this system level, each object can then be refined into component objects on a lower level object diagram, designed to meet the specification for the object. The result is a leveled set of object diagrams which completely describe the structure of a system. At the lowest level, objects are completely decomposed into primitive objects such as procedures and internal state data stores. At higher levels, object diagram leveling can be used in a manner similar to Booch's "subsystems".[6]

The seniority hierarchy is expressed by the topology of connections on a single object diagram (see figure 4). An arrow between objects indicates that one object calls one or more of the operations provided by another object. Any layer in a seniority hierarchy can call on any operation in junior layers, but never any operation in a senior layer. Thus, all cyclic relationships between objects must be contained within a virtual machine layer. Object diagrams are drawn with the seniority hierarchy shown vertically. Each senior object can be designed as if the operations provided by junior layers were "primitive operations" in an extended language. Each virtual machine layer will generally contain several objects, each designed according to the principles of abstraction and information hiding.

Designing Systems

The main advantage of a seniority hierarchy is that it reduces the coupling between objects. This is because all objects in one virtual machine layer need to know nothing about senior layers. Further, the centralization of the procedural and data flow control in senior objects can make a system easier to understand and modify.

However, this very centralization can cause a messy bottleneck. In such cases, the complexity of senior levels can be traded off against the coupling of junior levels. The important point is that the strength of the seniority hierarchy in a design can be chosen from a spectrum of possibilities, with the best design generally lying between the extremes. This gives the designer great power and flexibility in adapting system designs to specific applications.

Figure 5 shows one possible design for the ORBIT SIMULATOR. Note that, by convention, the arrow labeled "RUN" is the initial invocation of the entire system. In transitional design diagrams such as figure 5, it is sometimes convenient to show what data flows along certain control arrows, much in the manner of structure charts[23] or "Buhr charts"[7]. These annotations will not appear on the final object diagrams.

In figure 5, the junior level components do not interact directly. All data flow between junior level objects must pass through the senior object, though each object still receives and produces all specified data (for simplicity not all data flow is shown in figure 5). This design is somewhat like an object-oriented version of the structured designs of Yourdon and Constantine.[25]

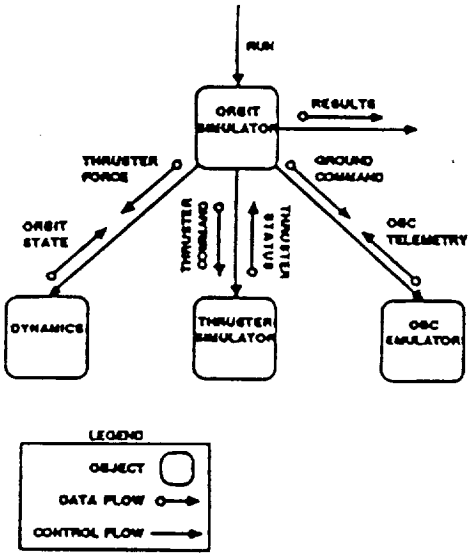

FIGURE 6 Orbit Simulator – Centralized Design

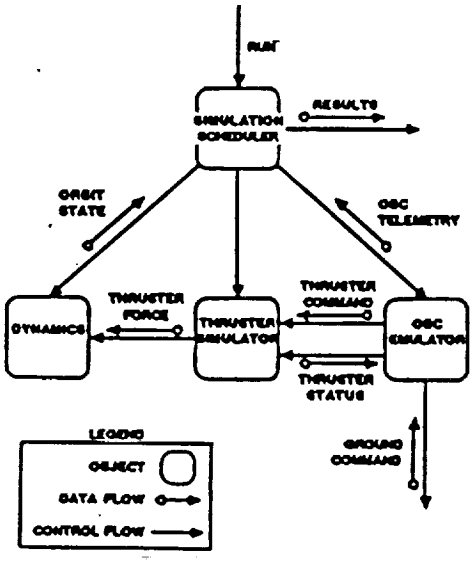

FIGURE 6 Orbit Simulator with Decentralized Data Flow

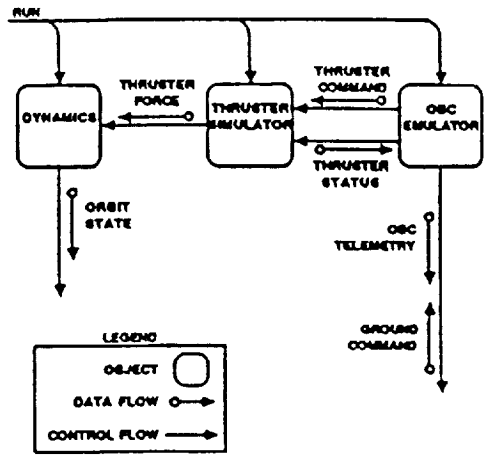

FIGURE 7 Orbit Simulator – Decentralized Design

We can remove the data flow control from the senior object and let the junior objects pass data directly between themselves, using operations within the virtual machine layer (see figure 6). The senior object has been reduced to simply activating various operations in the virtual machine layer, with very little data flow.

We can even remove the senior object completely by distributing control among the junior level objects (see figure 7). The splitting of the RUN control arrow in figure 7 means that the three objects are activated simultaneously and that they run concurrently. The seniority hierarchy has collapsed, leaving a homologous or non-hierarchical design[25] (no seniority hierarchy, that is; the composition hierarchy still remains).

A design which is decentralized like figure 7 at all composition levels is very similar to what would be produced by the PAMELA (tm) methodology.[10] In fact, it should be possible to apply PAMELA design criteria to the upper levels of an object diagram based design of a highly concurrent system. All concurrent objects would then be composed, at a certain level, of objects representing certain process "idioms".[10] Below this level concurrency would generally no longer be advantageous.

The entity-relationship model provides a basis for the data flowing on an ODD. Not all these entities are represented by specification objects, but they are generally at too high a level of abstraction to be directly represented by basic data structures. Therefore we need to add a virtual machine layer of objects to provide abstract data types which preserve the abstraction of these problem domain entities. In the case of the ORBIT SIMULATOR these data types might include VECTOR, MATRIX, GROUND COMMAND and simulation PARAMETER types. Figure 8 shows how these objects might be added to the simulator design of Figure 6.

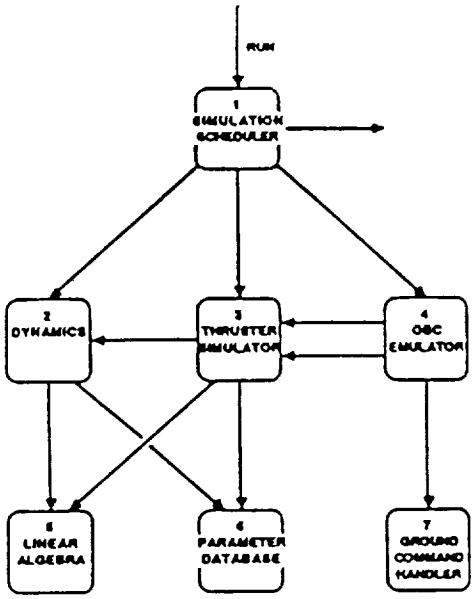

FIGURE 8 Orbit Simulator Design

Figure 8 gives one complete level of the design of the ORBIT SIMULATOR. Note that figure 8 does not include the data flow arrows used in earlier figures. When there are several control paths on a complicated object diagram, it rapidly becomes cumbersome to show data flows. Instead, object descriptions for each object on a diagram provide details of the data flow.

An object description includes a list of all operations provides by an object and, for each arrow leaving the object, a list of operations used from another object. We can identify the operations provided and used by each object in terms of the specified data flow and the designed control flow. The object description can be produced by matching data flows to operations. For example, the description for the DYNAMICS object might be:

Provides:
Initialize ()
Integrate (TIME INTERVAL)
Apply Thrust (THRUSTER FORCE)
Current State () ORBIT STATE

Uses:
5.0 LINEAR ALGEBRA
Vector Add
Dot Product
Scalar Multiply
Matrix Multiply

6.0 PARAMETER DATABASE
Get Dynamics Parameters

Data in parentheses are arguments which flow along the control arrows, while unparenthesized data are results which are returned.

For objects with specifications and lower level ODDs, we can recursively construct lower level object diagrams. These lower level designs must, however, both meet the functionality of the specification and provide the operations listed in the object description. Some design objects, however, will not have object specifications. Abstract data type objects can have their design based on the structure of the entities they represent. For "executive" objects like SIMULATION SCHEDULER it may be worth creating a specification for it before proceeding with design as above. In all cases, the design process continues recursively until the entire specification has been covered by the design and all objects are completely decomposed.

Implementation

The transition from an object diagram to Ada is straightforward. The relationship between object diagram notations and Ada language features is:

Object Diagram	Ada Construct
Object	Package
Procedure	Subprogram
State	Package/task variables
Arrow	Subprogram/entry call
Actor	Entries/Accepts (not covered in this paper)

Package specifications are derived from the list of operations provided by an object. For the DYNAMICS object from the last section the package specification is:

```
package Dynamics is

  type ORBIT_STATE is
    record
      Position : Linear_Algebra.VECTOR;
      Velocity : Linear_Algebra.VECTOR;
    end record;

  procedure Initialize;
  procedure Integrate
    ( For_Duration : in DURATION );
  procedure Apply_Thrust
    ( Force : in Linear_Algebra.VECTOR );
  function Current_State
    return ORBIT_STATE;

end Dynamics;
```

The package specifications derived from the top level object diagram can either be made library units or placed in the declarative part of the top level Ada procedure. For lower level object diagrams the mapping is similar, with component package specifications being nested in the package body of the composite object. States are mapped into package body variables. This direct mapping produces a highly nested program structure. Alternatively, some or all of these packages can be made library units or even reused from an existing library. However, this may require additional packages to contain data types and state variables used by two or more library units.

The process of transforming object diagrams to Ada is followed down all the object diagram levels until we reach the level of implementing individual subprograms. Low-level subprograms can be designed and implemented using traditional functional techniques. They should generally be coded as subunits, rather than being embedded in package bodies.

The clear definition of abstract interfaces in an object-oriented design can also greatly simplify testing. When testing an object, there is a well defined "virtual machine" of operations it requires from objects at a junior level of abstraction, some of which may be stubbed-out for initial testing. Further, object-oriented composition encourages incremental integration testing, since the "unit testing" of a composite object really consists of "integration testing" the component objects at a lower level of abstraction.

Reusability

Software reusability is one of the major drivers for the development of the Ada programming language. Ada features such as generic packages are useful tools, but language features are not sufficient to guarantee high levels of software reuse. What is also needed is an approach to specifying and designing reusable components. This section shows how our method supports such an approach, and also presents an example of how generic program units can be used in the context of the method.

Software reusability is still more of a research topic than part of standard practice in the field. This section will discuss three concepts that support a high level of reuse. They are to use an object-oriented approach, to reuse the products of all life cycle phases, and to provide documentation that is both useful and maintainable. This is by no means a complete list of relevant factors, but our experiences in developing flight dynamics software have proven their utility.

Object-Oriented Approach

Using an object-oriented approach is useful not because object-oriented design is essential for reuse, but because the underlying concepts are. The common elements that are important are:

-- Data abstraction and information hiding
-- Levels of virtual machines
-- Inheritance

Parnas[20] discusses the importance of the abstraction, information hiding, and virtual machine levels in making software easier to reuse. Cox shows the importance of inheritance by comparing the size of the Smalltalk environment (40,000 lines) and the Berkeley Unix environment (400,000 lines), where the former environment relied heavily on inheritance to promote reusability.[11]

Ada cannot support inheritance as easily as Smalltalk, but can simulate it through the use of the composition hierarchy. We will continue the sample orbit problem to show the simulation of inheritance and to show how Ada generics can be used in the context of object-oriented design.

In the object diagram for the orbit simulation system (figure 8) the object LINEAR ALGEBRA is at the lowest virtual machine level. It provides an extended language that allows the developers of objects such as DYNAMICS to write code in terms of linear algebra operations, rather than in terms of arrays and loops. A non-generic Ada library package can serve this purpose, but implementing a generic provides the advantage of controlling floating point accuracy.

Another reason to make a package generic is to ease the simulation of inheritance. We will demonstrate this by building the Dynamics package around a generic numeric integrator with the following specification:

```
generic

 type REAL is digits <>;
 type STATE_VECTOR is
  array (INTEGER range <>) of REAL;
 with function State_Derivative
  ( T : DURATION; -- from reference time
    X : STATE_VECTOR )
  return STATE_VECTOR;

package Generic_Integrator is

 procedure Integrate
  ( For_Duration : in DURATION );
 function Current_State
  return STATE_VECTOR;
 procedure Initialize ...;

end Generic_Integrator;
```

This package provides the ability to numerically integrate a vector differential equation with an arbitrary state vector size. The Integrate procedure can be implemented as a vector equation, or as a set of individual real-valued functions. To implement it as a single vector equation we will need the operations provided by package Linear_Algebra. These operations can be incorporated in two ways. One possibility is to make the operations needed into generic formal parameters. Another is to instantiate Linear_Algebra within the integrator itself. Each method has advantages and drawbacks. Using generic formal subprograms enhances reusability by making the component self-contained, but if too many are needed the interface becomes complex. Instantiating Linear_Algebra within the Generic_Integrator makes a cleaner interface, but couples the generic package to another library unit. The pilot project team has used both methods. Figure 9 shows how the object Integrator_Linear_Algebra is the instantiation of a generic package.

Figure 10 shows the composition of the DYNAMICS object. Orbit_Integrator is the instantiation of the generic package discussed above. The generic package is instantiated with Orbit_Equation as the actual parameter corresponding to the formal parameter function State_Derivative. The operations "Initialize" and "Apply Thrust" are shown in figure 10 as component procedures, represented by rectangles. The other Dynamics operations are "Current State" and "Integrate." These operations are inherited from the Orbit_Integrator object.

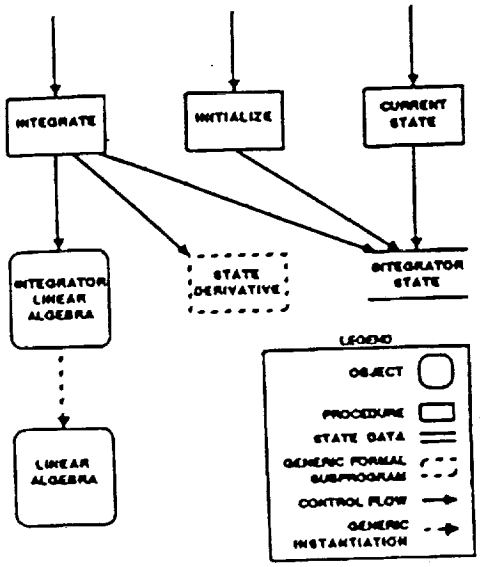

FIGURE 9 Generic Integrator

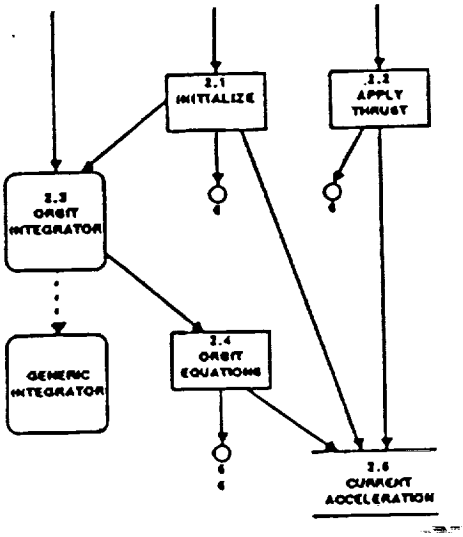

FIGURE 10 Dynamics

Smalltalk's subclassing[13] provides an elegant means of supporting inheritance. Ada does not directly support inheritance, but the concept can be simulated by using "call-throughs." A call-through is a subprogram that has little function other than to call on another package's subprogram.

To simulate inheritance when implementing the Dynamics package the subprograms Integrate and Current_State would be respecified in the Dynamics package, with the subprogram bodies in Dynamics calling on the corresponding operations in Orbit_Integrator. The call-through for Current_State would also have to take care of converting the lower-level data type "STATE_VECTOR" returned by the Integrator to the higher-level type "ORBIT_STATE".

This technique is clearly less elegant than Smalltalk subclassing, but it also has advantages. First, Ada allows inheritance from more than one object. Secondly, Smalltalk forces the inheritance of all operations and data. An operation can be overridden, but not removed, from a class. The Ada specification of the composite package gives the developer precise control over which operations and data items are visible or accessible. In the Dynamics example the operations "Integrate" and "Current_State" are inherited by Dynamics, but "Initialize" is not.

Reuse Across the Life-Cycle

Reuse across the life cycle is a concept promoted in both research[16] and production environments. Toshiba claims that the life cycle approach used in their software factory yields 85 percent software reuse[15]. The key to Toshiba's model is that the life cycle is represented as a series of transformations from a user need to the final product, thus preserving traceability. The final software is made more general (and thus reusable), then the specification is rewritten into a "presentation" that is consistent with the generalized code. These presentations are used when new system requirements are being developed.

Our method can be used to support a model such as Toshiba's, because objects can be traced from the ODDs to object diagrams to Ada code. Toshiba's concepts need to be refined so that a single requirement (eg., "integrate orbit equation") can be mapped into several implementations (eg., different numerical integration algorithms). Another drawback of the Toshiba model is that it is not designed to handle a wide variety of problem domains. We address this by leaving room for application-dependent notations in the object specification, and by providing the two orthogonal design hierarchies.

The specification and design documents must be maintained along with the code. These documents provide the traceability that is needed for software reuse. Our pilot project team has found that the volume of documentation generated makes it hard to keep the design notebook consistent with the source code. Adding a specification notebook will compound this problem. The solution is to maintain as much information as possible on a computer, and to extend the use of configuration control software to the specification and design.

Another factor in reusing software is how the documentation should be accessed by developers when there is a large library of reusable components. The similarity between the entity dictionary and AI knowledge representations encourages the belief that expert systems may ultimately play a role. However, more conventional library tools have the advantage of not being tied to the development method used. Intermetrics has been doing research into such systems[8] by developing a prototype Ada Software Catalog (ASCAT).

Conclusion

The techniques described in this paper have evolved from our experience with two Ada projects and one Modula-2 project in addition to the original pilot project. As of this writing all these projects are in the late stages of development. The response to the object-oriented approach has generally been quite favorable, once the new techniques are understood.

As these projects are completed, we plan to use the substantial amount of data being gathered to begin to quantify the productivity of our methods. All the systems will be used in actual operational environments, allowing us to study their reliability and maintainability.

The traditional functional viewpoint provides a comprehensive framework for the entire software life-cycle. This viewpoint reflects the action-oriented nature of the machines on which software is run. As we have discussed here, the object-oriented approach can also provide a comprehensive view of the life-cycle. The object-oriented viewpoint, however, reflects the natural structure of the problem domain rather than the implicit structure of our hardware. Thus, it provides a "higher-level" approach to software development which decreases the distance between problem domain and software solution. By making complex software easier to understand, this simplifies both the system development and maintenance. This is the goal of our general object-oriented Ada life-cycle.

References

1. Agresti, William W. "An Approach to Developing Specification Measures," Proceedings of the 9th Annual Software Engineering Workshop, GSFC Document SEL-84-004, November 1984.

2. Agresti, William W., et. al. "Designing with Ada for Satellite Simulation: a Case Study," Proceedings of the 1st International Conference on Ada Applications for the Space Station, June 1986.

3. Basili, V. R., et. al. "Characterization of an Ada Software Development," Computer, September 1985.

4. Booch, Grady. Software Engineering with Ada, Benjamin/Cummings, 1983.

5. Booch, Grady. "Object-Oriented Software Development," IEEE Transactions on Software Engineering, February 1986.

6. Booch, Grady. Software Components with Ada, Benjamin/Cummings, 1987.

7. Buhr, R. J. A. System Design with Ada, Prentice-Hall, 1984.

8. Burton, Bruce and Micheal Broido. "Development of an Ada Package Library", Proceedings of the 1st International Conference on Ada Applications for the Space Station, June 1986.

9. Chen, P. "The Entity-Relationship Model -- Toward a Unified View of Data," ACM Transactions on Data Base Systems, March 1976.

10. Cherry, George W. PAMELA Designer's Handbook, Thought**Tools, 1986.

11. Cox, Brad. "Message Object Programming: An Evolutionary Change in Programming Technology," IEEE Software, January 1984.

12. Dijkstra, Edsger W. "The Structure of the 'THE' Multiprogramming System," Communications of the ACM, May 1968.

13. Goldberg, Adele and David Robson. Smalltalk-80: The Language and Its Implementation, Addison-Wesley, 1983.

14. Martin, James and Carma McClure. Diagramming Techniques for Analysts and Programmers, Prentice-Hall, 1985.

15. Matsumoto, Yoshihiro. "Some Experience in Promoting Reusable Software: Presentation in Higher Abstract Levels," IEEE Transactions on Software Engineering, September 1984.

16. McKay, Charles. Lecture to GSFC Ada Users Group, April,1986.

17. Minsky, Marvin. "A Framework for Representing Knowledge," in The Psychology of Computer Vision, ed. by P. Winston, McGraw-Hill, 1975.

18. Nelson, Robert W. "NASA Ada Experiment -- Attitude Dynamic Simulator," Proceedings of the Washington Ada Symposium, March 1986.

19. Parnas, David L. "On the Criteria to be Used in Decomposing Systems into Modules," Communications of the ACM, December, 1972.

20. Parnas, David L. "Designing Software for Ease of Expansion and Contraction," IEEE Transactions on Software Engineering, March 1979.

21. Quillian, M. R. "Semantic Memory" in Semantic Information Processing, ed. by M. Minsky, MIT Press, 1968.

22. Rajlich, Vaclav. "Paradigms for Design and Implementation in Ada," Communications of the ACM, July 1985.

23. Seidewitz, Ed and Mike Stark. "Towards a General Object-Oriented Software Development Methodology," Proceedings of the 1st International Conference on Ada Applications for the Space Station, June 1986.

24. Seidewitz, Ed and Mike Stark. General Object-Oriented Software Development, GSFC Document SEL-86-002, August 1986.

25. Yourdon, Edward and Larry L. Constantine. Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design, Yourdon Press, 1978.

Ed Seidewitz works at the NASA Goddard Space Flight Center as a flight dynamics analyst. He is also very involved in the development of analysis software, user interface design and applications of Ada. He has previously worked in space systems engineering, computer aided instruction and artificial intelligence. He holds two B.S. degrees from the Massachusetts Institute of Technology, one in Aeronautics and Astronautics and one in Computer Science and Engineering. He is a member of the American Institute for Aeronautics and Astronautics and the Association of Computing Machinery.

Mike Stark works in the Systems Development Branch at the NASA Goddard Space Flight Center. He has worked on flight dynamics support software for the ERBS and COBE satellites and is currently involved in the implementation of an attitude dynamics simulator in Ada. He holds a B.A. degree in Mathematics and Economics from Oberlin College, and is currently working on a Masters in Computer Science in the Johns Hopkins University Part Time Engineering Program.